LEAGUE OF AMAZING PROGRAMMERS

LEVEL 3: CHEAT GUIDE

CREATED BY STUDENTS:

ATHENA HERNANDEZ
CINDY FELDMAN
ELLA DEMAREST
RYLAND BIRCHMEIER
SHIVA KANSAGARA
SOFIA VELARDE

TABLE OF CONTENTS

ORGANIZED BY MODULE

Module 0: Array and 2D Array

- **Array:** An object that can hold a given number of a single type of variable.
  Declaration:

  type[] arr;

  Example: int[] arr;

  *always include the brackets after either the type or after the variable name.

  Initializing:

  type[] arr = new type[size];

  *size is the number of elements in the array ‘arr’

  Example: int[] arr = new arr[5];

  Access to array elements:
  - An array can have any positive integer number of elements in it.
    Access element: type[] arr = new type[size]

    arr[0] - accesses the first element in array ‘arr’.

    Example: int[] arr = new arr[5]; arr[0] =1; this sets the integer at element 0 in the array to 1.

- Continue with "array iteration" on next page...

Array iteration:

- To access every element in an array or iterate through an array, you use a loop.
- This is helpful for initializing elements in said array.
- **arr.length** is used to get the number of elements in the array. Useful for array iteration.

- <u>Example:</u>

```
int arr[] = new int[5];
for(int i = 0; i < arr.length;i++){
      arr[i] = 5;
      System.out.println(arr[i]);
}
```

*this sets every element in the arr array to 5 and then prints out 5 fives

Output:

```
      5
      5
      5
      5
      5
```

2D Array:

- Like a 1D array, a 2D array is declared with its type and variable name.
- This time you need two brackets instead of one, hence the 2D.

type[][] twoDArray = new type[rows][columns]

  - variable name
  - # of rows
  - # of columns

<u>Example:</u>

int[][] arr = new int[5][6];

*this initializes a 2D array with 5 rows and 6 columns named arr.

A **nested for-loop** is used to iterate through 2D arrays.

*see level 0 module 5 for in-depth explanation of nested for- loops.

<u>Module 1: ArrayList and HashMap</u>

ArrayLists:

- Personally, I use ArrayLists probably more than I should as an alternative to other data structures. They are a very capable structure with so many methods, it is hard to resist.

- Creating ArrayLists:

  ArrayList<String> strings = new ArrayList<String>();

- Two things to note:
  - Have to still put the parenthesis even though you aren't usually going to put something in them (you definitely can though, but that is for another time).
  - The second <String> is optional, so you can write it like this as well and nothing changes...

  ArrayList<String> strings = new ArrayList<>();

- Subtle difference, yet good to know.

- There are too many ArrayList methods to outline here, but here are just a few of the more useful ones:

  .add(E element); / .add(int index, E element);
  .remove(int index).set(int index, E newElement);
  .isEmpty()
  .size();.
  clear();
  .contains(Object o);

HashMaps:

- A **hashmap** is useful to put two values together with a key and a value.
- Here is a great example of this in action:

  ```
  HashMap<String, String> capitalCities = new
  HashMap<String, String>();
  ```

- We can also add to this list very easily with the .put() method...

  ```
  capitalCities.put("England", "London");
  capitalCities.put("Germany", "Berlin");
  capitalCities.put("Norway", "Oslo");
  capitalCities.put("USA", "Washington DC");
  ```

- If we want to retrieve the capital city for a country now, all we have to do is use the .get() method.
- <u>Example</u>:

  ```
      capitalCities.get("England");
  ```

- You can mix data types and do a lot more. Many of the same ArrayList methods apply here as well, though by no means all of them.

Module 2: Search and Sort Algorithms

Sorting:

- Sorting is a very important aspect of computing. There are many different approaches to sorting that are efficient with different sets of data.
- Here are just a few examples and summaries of ways to sort:
    - **Insertion Sort**: putting an element in the appropriate place in a sorted list yields a larger sorted list.
    - **Bubble Sort**: rearrange pairs of elements which are out of order, until no such pairs remain.
    - **Selection Sort**: extract the largest element from the list, exchange with the last element in the current list, and repeat.
    - **Bucket Sort**: separate into piles based on the first letter, then sort each pile.
    - **Merge Sort**: Two sorted lists can be easily combined to form a sorted list.
- In this module the SortingVisualizer will help you better understand how each algorithm works, and give you a better idea of how to approach coding each one.

Searching:

- Like sorting, there can be different algorithms used to search a list.
- For example, algorithms to search depend on if the list is sorted or not.

- **Linear Search**: looks at the first list item to see whether you are searching for it and, if so, you are finished. If not, it looks at the next item and on through each entry in the list.

- **Binary Search**: starts at the middle of the database. If your target number is greater than the middle number, the search will continue with the upper half of the list. If your target number is smaller than the middle number, the search will continue with the lower half of the list. It keeps repeating this process, cutting the list in half each time until it finds the record.

<u>Module 3: String Methods and Algorithms</u>

Working with **Strings**:

- You have already done a lot with Strings before, but learning how to manipulate and build them in more complex ways is very useful!

- Here are some reminders of methods you can use with Strings and chars:

    - **exampleString.charAt(indexOfCharacter):** returns the character at a specified location in the
    - **StringexampleString.toLowerCase()**: returns the String in all lowercase characters
    - **exampleString.toUpperCase()**: returns the String in all uppercase
    - **charactersCharacter.toString(exampleString.charAt(indexOfCharacter)**: Converts and returns a specified character to a String
    - **exampleString.substring(startIndex, endIndex)**: returns a portion of a String from the specified start and end index parameters

Summary of **Abstract Classes**:

- An abstract class is a restricted class that cannot be used to create objects. Instead, it must be inherited from another class.
- You will be using animals to illustrate this concept. A living thing cannot be specifically identified as just an “animal”; it is a species and has attributes that make it unique. For example, a pig is still an animal, but when identifying it from other animals, it must be specified that it is a pig.
- Simple Example of an Abstract Class
    - (Note: the methods in abstract classes are also abstract, and are not defined. You must define their function in the classes that extend the abstract class.)

```
abstract class Animal {
    abstract void makeNoise();
    abstract void doTrick();
}
```

Module 4: Stack and Queue

Stack Summary:

- A stack is sort of like an array list in that it stores a set of objects. The main difference here is the order that a stack stores the objects.
- A stack has two main methods: push() and pop(). You cannot add or remove an object from the middle of the stack.

Stack Declaration:

Stack<var type> stackName = new Stack<var type>();

Example: Stack<String> stackOfStrings = new Stack<String>();
*This declares and initializes a stack called stackOfStrings

Stack push() and pop():

- **push()**: is used to add an object to the stack. It is pushed to the top of the stack.

  Example:

  ```
  stackOfStrings.push("hello");
  stackOfStrings.push("world");
  ```

  *Pushed “hello” and “world” to the stack.

- **pop()**: is used to remove an object from the stack. It removes the top-most element.

  Example: stackOfStrings.pop();
  *removed “world” from the stack because it is the topmost element.

- **Removing all objects from a Stack**:

  ```
  while(!stackOfStrings.isEmpty()){
        System.out.println(stackOfStrings.pop());
  }
  ```

  *This pops all the strings off the stack, leaving it empty. It will print out the strings in the order removed.

  ```
  Here it prints: world
                  hello
  ```

- **Queue summary**: Queue is similar to stack in that it has a set of objects that can only be used in a specific order. Instead of starting from the topmost element, a queue starts from the oldest object or bottom most element. The oldest object in a queue is called the front/head and the newest is called the back/tail.

- **Queue declaration:**

  ArrayDeque<String> queueOfStrings= new ArrayDeque<String>();

  *this declares and initializes a queue called queueOfStrings

- **Queue add and remove:**

  add(): queueOfStrings.add("hello");
  queueOfStrings.add("my");
  queueOfStrings.add("friend");

  *This adds the three strings to the queueOfStrings Queue.

  remove(): queueOfStrings.remove();

  *This gets rid of the "hello" string since it is the "oldest" in the Queue.

<u>Module 5 - Intro to Recursion:</u>

- **Recursion**: the process when a method calls itself continuously.
- A method that calls itself is **recursive.**

  Format:

```
returntype methodname(){
      //code to be executed
      methodname(); //calling same method
}
```

- Recursion usually results in a StackOverflowError
- Example of recursion that happens **infinite** times:

```
public class RecursionExample1 {
    static void p(){
        System.out.println("hello");
        p();
    }

    public static void main(String[] args) {
        p();
    }
}
```

Output:

```
    hello
    hello
    ...
    java.lang.StackOverflowError
```

- Example of recursion that happens a **finite** amount of times:

```
public class RecursionExample2 {
    static int count=0;
    static void p(){
        count++;
        if(count<=5){
            System.out.println("hello "+count);
            p();
        }
    }

    public static void main(String[] args) {
        p();
    }
}
```

Output:

```
    hello 1
    hello 2
    hello 3
    hello 4
    hello 5
```

- How recursion is used is up to the program that is being developed.
    - Things such as factorial numbers and the Fibonacci Sequence can be coded using recursion.